20030128003

2

Technical Memorandum
SEI-86-TM-14

Carnegie-Mellon University
Software Engineering Institute

Intelligent Assistance without Artificial Intelligence

by

Gail E. Kaiser

and

Peter H. Feiler

September 1986

2

Technical Memorandum
SEI-86-TM-14
September 1986

Intelligent Assistance without Artificial Intelligence

by

Gail E. Kaiser[*]
Columbia University

and

Peter H. Feiler
Software Engineering Institute

*This paper was written while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA 15213.

The development and maintenance of SMILE is supported in part by the United States Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, NJ and in part by ZTI-SOF of Siemens AG, Munich Germany.

Table of Contents

List of Figures

Intelligent Assistance without Artificial Intelligence

Gail E. Kaiser[1]

and

Peter H. Feller

ABSTRACT. SMILE is a distributed, multi-user software engineering environment that behaves as an intelligent assistant. SMILE presents a 'fileless environment', derives and transforms data to shelter users from entering redundant information, automatically invokes programming tools, and actively participates in the software development and maintenance process. Unlike other intelligent assistants, SMILE is not a rule-based environment: its knowledge of software objects and the programming process is hardcoded into the environment. We describe SMILE's functionality and explain how we achieved this functionality without reliance on artificial intelligence technology.

1 Introduction

In 1973, Winograd discussed his dream of an intelligent assistant for programmers [29]. More recently, artificial intelligence researchers have extended programming languages and environments (primarily Lisp environments) with knowledge about the relationships among program units [26] and the rules governing the software development process [3, 1, 22] in an attempt to turn the dream into reality. The resulting systems support 'exploratory programming' by an individual programmer very well [21], but they do not provide the assistance necessary to manage large-scale development and maintenance. However, as AI projects, such as 'expert systems', have become larger and commercially viable, researchers have turned their efforts towards developing this kind of assistance [11, 18], and we believe they will produce excellent results.

[illegible]

In the meantime, it is possible to build production-quality software engineering environments that provide seemingly intelligent assistance without requiring new breakthroughs in AI research. There is already (at least) one such system—the Software Management and Incremental Language Editing system (SMILE)—that provides seemingly intelligent, interactive support for teams of software developers and maintainers. SMILE does not use artificial intelligence techniques; it is not even written in Lisp. SMILE was written in C and runs on Unix™.

Although SMILE is several years old, it has not been discussed in the literature, except in acknowledgements by researchers who used it to develop their own systems. SMILE was developed by one of the authors, starting in 1979, originally as a tool for developing research prototypes for the Gandalf project [16]; it has been used extensively by both authors and by many others since 1980. SMILE has been relied on by the Gandalf and Gnome [7] projects at CMU and by the Inscape project [17] at AT&T Bell Labs; it has been distributed to at least forty sites. SMILE

[1]This paper was written while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

passes the crucial test of supporting its own maintenance. The purpose of this paper is to present the goals of SMILE and explain how they were achieved.

The original, high-level goals of SMILE were as follows.

- To hide the file system and the operating system from the users. SMILE presents a 'fileless environment'; that is, SMILE exposes its users only to the logical structure of the target software system. The normal alternative is for users to deal with the physical storage of the software in terms of directories and files, which often do not correspond nicely to the logical structure.
- To shelter the users from the tedious task of maintaining redundant information. SMILE requires its users to enter each item of information only once; it automatically transforms the data as needed by tools. SMILE derives necessary information that can be calculated from the data supplied by users.
- To automate invocation of tools at appropriate points. SMILE assists the users by automatically performing trivial software development activities such as calling grep, lint, cc, make, and other Unix utilities [12] with the appropriate arguments at appropriate times. In some cases, the tool is invoked as soon as its input is ready; in other cases, the tool is not called until its results are required, such as to answer a user query or to provide input to another tool. SMILE hides the particularities of Unix and presents a uniform programming model different from the model imposed by Unix.
- To actively participate in the software development and maintenance process. SMILE is an interactive system, and all programming activities take place within the environment. In addition to calculating auxiliary information and automatically invoking tools, SMILE anticipates the consequences of user actions and automatically presents appropriate warning messages.
- To be sufficiently robust and reliable for supporting relatively large academic development projects. It automatically recovers from inconsistent states after user-initiated aborts and machine crashes; it also stores information redundantly to support recovery from disk errors or its own bugs.

All of these goals have been achieved. SMILE maintains source code, object code and other software development information in a database mapped onto the Unix file system. Knowledge of software objects and a model of the software development process are hardcoded into SMILE's commands. SMILE incorporates a large collection of Unix utilities, plus several special tools developed as part of the Gandalf research. SMILE has supported the simultaneous activities of at least seven programmers, and the largest software system developed and maintained in SMILE has approximately 61,000 lines of source code [13].

The following sections present the goals and achievements of SMILE in more detail. Section 2 explains SMILE's external architecture. Section 3 describes how SMILE assists individual programmers, while Section 4 describes the facilities oriented towards projects involving many programmers and long lifetimes. Section 5 discusses SMILE's implementation and current status. Section 6 compares SMILE to other software engineering environments. We conclude by summarizing the significance of SMILE as an example of intelligent assistance without artificial intelligence.

2 Architecture

SMILE is intended for use by small teams of programmers (5 to 20) developing and maintaining medium-size software systems (10,000 to 250,000 lines of source code) written in C, taking maximum advantage of the Unix file system and utilities.

2.1 GC

GC (Gandalf C) [28] is an enhancement of C that lists the types of formal parameters within the argument list (as in Pascal) and provides a module interconnection language (MIL). The MIL defines modules consisting of four types of source code objects (called *items*): procedures, variables, types, and macros. Each module has an import list indicating the items required from other modules and an export list indicating the items accessible to other modules. GC was adopted by the Gandalf project for all implementation efforts. SMILE supports GC, but automatically transforms source and header files from standard C to GC and *vice versa* as needed to import existing source code and to take advantage of C-specific programming tools. Throughout the rest of this paper, we mean "GC" when we say "C".

2.2 Databases

SMILE maintains all information about a software system in a *database* similar to the 'objectbases' of more recently developed programming environments [2, 15]. Each object has several attributes, representing auxiliary information, and is typed, enabling SMILE to provide object-oriented commands that apply type-specific tools.

A database consists of one or more 'projects', each representing a distinct software system. Most databases contain exactly one project, so we say 'database' and 'project' interchangeably. A project contains a number of 'modules' corresponding to GC modules. Each module contains a set of procedures, a set of variables, a set of type definitions, a set of macros, a list of import items, and a list of export items, as illustrated in Figure 1. The source text of procedures, variables, types, and macros are written in GC. Each module and item is attributed with status information, such as whether or not it has been compiled since it was last modified. Modules also contain object code, but this is never explicitly visible to users.

2.3 User Interface

SMILE's user interface is script-oriented, and does not take advantage of windows or menus.[2] However, some tools included in SMILE, *e.g.*, screen-oriented editors, behave differently for bit-mapped screens than for dumb terminals.

[2]The workstation implementations of SMILE do support windows. In particular, a user can modify a database in one window while browsing through the database in a second, read-only window; see Section 5.

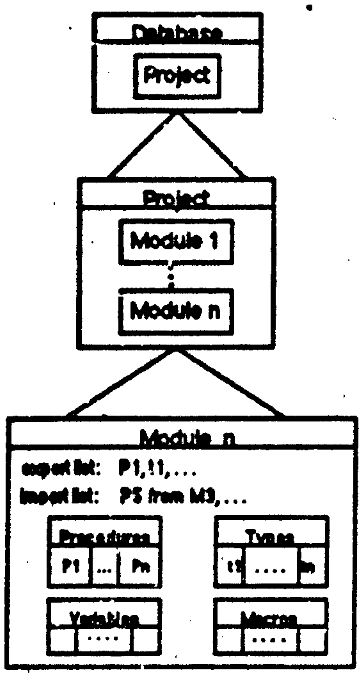

Figure 1: Database of Software Objects

The user interface is 'friendly' and includes on-line help facilities. It is not necessary to remember either commands or arguments. The user can type a carriage-return after entering any part of the command line, and SMILE will prompt, one at a time, for remaining arguments; each prompt indicates a default value based on the user's most recent activities, and the user types a carriage-return to accept this default. If the user types "?" at any point, SMILE lists the currently valid alternatives according to the user's context. If the user instead enters "help", then SMILE explains the selected command and its argument. The user can also hit the interrupt key to any prompt to abort the current command. SMILE permits the user to abbreviate commands and arguments of commands to the shortest unambiguous form, and prompts with the possible choices if an abbreviation is ambiguous.

3 Programming Assistance

SMILE assists individuals in writing programs. It maintains C source code, object code, and the status of these objects in its database, and automatically performs menial development activities. For example, it warns the programmer of the implications of changing the specifications of source code items, and it automatically recompiles after changes.

3.1 Browsing

SMILE helps the user navigate through a software system. The user selects a particular module—the user's *focus*—which is then indicated in SMILE's top-level prompt. SMILE assumes that further commands refer to this module and its contents until a different module is selected.

Browsing is object-oriented, in the sense that SMILE automatically invokes the appropriate viewing tool according to the type of the selected object. For C source code, this is normally a screen-oriented text editor; an earlier version of SMILE also provided a syntax-directed editor. Although SMILE assumes all commands are with respect to the current focus, it can shift focus automatically as the need arises. For example, if the user asks to visit an item that is not in the current module but is in some other module, SMILE changes the focus before invoking the appropriate viewer tool.

SMILE also supports general searches. A query can apply to an individual item, a module, or the entire database and SMILE can further filter the results of queries to display only items of a particular type (import item, procedure, variable, *etc.*) or only items that match some pattern. Pattern matching can be applied to the name of an item or to its source text, and a search can be applied to either the definition or usage of items, or SMILE can generate a full cross-reference table. The results of queries are displayed on the screen in the form of a transcript, which can be scrolled if SMILE is run from within a text editor that supports user shells. SMILE can also direct its answers to an external file or a printer. SMILE remembers past activities on a user-by-user basis; this supports, for example, a special option for the printer to spool only those items that have changed since last printed by the particular user.

3.2 Editing

SMILE creates and deletes modules and items within modules. If the user asks to remove an item that is in another module, SMILE requests confirmation before automatically changing focus to the other module to carry out the command. Thus, SMILE is forgiving of minor user errors. The add command requires the type of the new item; if this is not given, SMILE prompts for the missing argument. SMILE invokes the type-specific tool, and the low-level commands provided by the tool are used to construct the content of the item. If the user enters a command to write, save, save-and-exit, *etc.*, then the new item is stored in the database; if the user tells the tool to abort or exit (without saving), *etc.*, SMILE aborts the original add command. SMILE does this by monitoring the tool; no changes to the tools themselves are required.

Similarly, an existing item can be added or removed from the imports or exports list of the current module. When a new item is created, SMILE automatically asks the user whether or not it should be added to the exports list. When an item is removed from the exports, SMILE warns the user if it is imported by other modules and requests confirmation; if confirmation is given, it automatically removes these imports as well. When a user tries to delete an existing item, SMILE reminds the user if it is exported and requests confirmation before removing the corresponding item from the export list.

The user can make changes to items through the edit and change commands; SMILE invokes the appropriate editing tool. Edit restricts the user to making local changes to the body of an item, whereas change allows the user to make changes to both the specification and the body, which may have side effects on other items. For example, edit invokes the editor tool only on the body of a C procedure; if the tool supports multiple windows, then the header of the procedure is displayed for reference in another, read-only window. In the case of a C variable, edit permits the user to modify the initialization, but not the actual declaration. The edit command does not apply to types and macros, because any modification can affect usages.

Sometimes changing the specification of an item has implications beyond those anticipated by the user. Therefore, SMILE always informs the user of potential problems before the damage is done. When the user selects the change command, SMILE queries its database to find all the other items that may be affected by the proposed change and informs the user of the extent of this change, in terms of how many other items might subsequently have to be modified to maintain consistency; it displays the actual dependencies on request. The user can abort or go ahead with the change with full knowledge of the implications.

3.3 Error Detection and Error Reporting

After a user adds, removes, or modifies an item, SMILE supplies rapid feedback regarding static semantic errors. The semantic analysis is applied only to the changed item rather than to other items affected by the change. SMILE propagates the change by updating the status information for dependent items. If the user requests it, SMILE submits these for analysis; this is explained in the following section.

The analysis is performed in a background process, so that the user does not have to wait for the tool to complete before continuing other activities. When processing completes, all error or warning messages are saved as an attribute of the current module (the focus), and the prompt is changed to indicate the errors. The user can ignore the errors, or ask SMILE to display the messages; thus, SMILE separates error detection from error reporting. Both the messages and the visual cue in the prompt remain until the user edits the offending item, so the user does not need to remember the particular errors or even the fact that there are errors within the particular module. It is less intrusive to indicate errors by appending a notice to the prompt than to display the errors themselves. An earlier version of SMILE dumped all the error messages on the user's screen as soon as the tool completed. This behavior was judged unacceptable because it interrupted the user's activities; the user was forced to read the messages then and remember them, because they were not stored.

3.4 Bookkeeping

SMILE maintains status information for each item. For example, each C item has a status field that indicates whether or not its static semantic analysis is up to date, whether the analysis was successful, or whether analysis is in progress in the background. SMILE maintains the correct value for the status field. Furthermore, SMILE automatically propagates changes to items by updating the status field of other items that might be affected by the change. The user can examine the status information for any item or display all items with a particular status. A user might use this information, for example, to request re-analysis of a particular item or all items affected by a change or to look for items that still have errors and need correction.

SMILE performs code generation by compiling at the granularity of a module. Therefore, it maintains a status field for each module indicating whether or not its object code is up to date, or being generated in the background. After compiling a module, SMILE indicates the resulting status in this status field. SMILE invalidates generated code by setting the status field accordingly under any one of several conditions:

- a new item is added to the module;
- an existing item is moved between modules, removed, edited or changed;
- an item is added to or removed from the import list, and this item is actually referenced by an item of the module;
- an exported item is changed, and this item is imported into another module, where it is actually referenced by an item in the importing module.

3.5 Code Generation and Linking

SMILE recompiles modules and relinks the system as needed. It recognizes several alternative notions of 'as needed'. There is a tradeoff between recompiling immediately after a item of a module changes and delaying until the user requests system execution: Late recompilation requires the user to wait, but early recompilation may be wasted due to further changes to the same module; it also affects response time after each change. An earlier version of SMILE automatically recompiled as soon as an item changed, but recompiled only the item rather than the entire module. This was changed because the time and space overhead was unacceptable. The processing performed by the compilation tool after every modification led to slower response due to the cycles taken by the background job. Space was a problem because a separate object code file was generated for each item. SMILE now compiles an entire module rather than individual items. This optimization was done without affecting the interaction with the user.

SMILE automatically generates a makefile, including the appropriate command lines, and invokes make to construct an executable system. If a file name is given as an argument, the executable code is placed in this file; otherwise, standard Unix practice is followed and the output goes to the "a.out" file in the current working directory.

3.6 Modes

Modes permit the user to control and adapt SMILE's behavior. Users can set modes explicitly with a command or implicitly in their SMILE *profiles*. Every mode has a type and a default value. The boolean Autocompilation mode permits the user to indicate to SMILE whether it should temporarily refrain from automatically carrying out analysis and code generation. This is a desirable feature when the user starts making major changes to the application. Another boolean mode related to compilation indicates whether or not the compiler should generate more elaborate debugging information. The Verbose mode indicates the level of verbosity of SMILE's warnings and suggestions.

Somes modes are used to tailor SMILE to a particular operating environment. CMU mode permits SMILE to take advantage of some special CMU utilities. Home mode defines SMILE's home directory in the local file system, and Print mode names the local tool for spooling to the printer. SMILE is also tailored by the search paths and other environment variables defined in the user's Unix profile.

4 Development and Maintenance Assistance

SMILE assists software teams with their long-term development and maintenance activities. It coordinates simultaneous activities by multiple users, encourages reuse of existing code, and logs source code changes.

4.1 Reservations

SMILE prevents multiple users from modifying the same module at the same time by requiring the user to **reserve** a module before making changes to the module. If a user tries to modify a component of a module that is not reserved, SMILE explains that reservation is necessary. Only one user can reserve a module at a time. If another user attempts to reserve a previously reserved module, SMILE informs the user about who has reserved the module; users can also query reservation status explicitly.

SMILE helps users avoid making incompatible changes. If a user tries to change the specification of an exported item, SMILE checks to make sure that that all the modules that import this item are also reserved by the same user. If not, SMILE informs the user of their reservation status.

4.2 Experimental Databases

Reservations are always made with respect to a private workspace called an *experimental database*. Figure 2 shows the relationship between experimental databases and the *public database*, which contains the baseline version of the software system. The modules in the public database are available to all members of the software team, while the contents of an experimental database are private to its owner. An experimental database is a logical copy of the public database; SMILE employs a copy-on-write strategy to conserve space. Only modules reserved in the current experimental database can be modified. Additional modules can be reserved at any time, provided they are not already reserved by another user. SMILE automatically prelinks non-reserved modules (in a background process) to improve the response time of system generation.

When a user completes a set of changes, the user gives either the **update** or **deposit** command

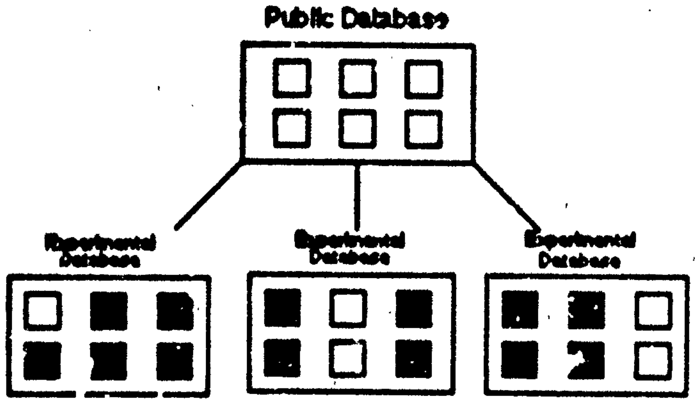

Figure 2: Experimental and Public Databases

to return all the reserved modules to the public database. Update retains the reservations, so the user can make further changes, while deposit removes the reservations. In either case, SMILE makes the changes available to the rest of the software team by replacing the previous versions in the public database with the changed modules from the experimental database. SMILE permits users to back out of a proposed change by releasing the current reservations, so other users can reserve these modules in their original state.

At the beginning of an update or a deposit, SMILE checks the status of all reserved items to ensure that they have been analyzed and compiled successfully, without any errors. If there are inconsistencies, SMILE aborts the command; otherwise, SMILE locks the public database while it copies the modified objects back into the public database. Thus, update and deposit behave as transactions with respect to the public database.

4.3 Transactions

Every SMILE command is a *transaction*, in the sense that it is impossible to apply a second command within the same database until the first command terminates.[3] Background processing is coordinated in such a way that its results are recorded without conflicting with the transaction model. An earlier version of SMILE saved its internal state on disk after each transaction in order to record the changes in a fail-safe manner. This led to poor responsiveness when there were five or more simultaneous users in a time-sharing environment (on a VAX™ 780) and was

[3]In the workstation implementations of SMILE, a user can access unaffected parts of a database in a read-only window during a transaction; see Section 5.

discontinued.[4] Currently, SMILE saves state after the number of transactions indicated by the Checkpoint mode, and always saves state before and after commands that cause major changes, such as change, update, and deposit. A user can select full state saving by setting the Checkpoint mode to 1; alternatively, the user can explicitly give the chkpoint command after particularly crucial changes.

SMILE coordinates changes among the experimental databases owned by the members of a software project. A user can add a new module only within an experimental database, but SMILE records the addition in the public database to prevent another user from adding another new module with the same name. Similarly, SMILE records addition of import items in the public database, since another user may attempt to delete the item in a different experimental database. When the public database is locked during a transaction, other actions that affect the public database are blocked until completion of the transaction. Since update and deposit often take several minutes, blocked commands time out after thirty seconds and SMILE advises the user to try again later. This enables users to perform other development activities while they wait.

4.4 Change Logs

When programming teams are large, it is useful to maintain on-line change logs. Whenever a user updates or deposits the contents of an experimental database, SMILE prompts for a log entry for each modified module. SMILE automatically includes the user's name, the time/date, and the module name with the text provided by the user. Users can also append log entries for their reserved modules at any time. A user can query the entire log for a database or only the log for a particular module, and request entries since a particular date and/or by a particular user. SMILE prevents tampering with previous log entries, so a full audit trail of past changes is always available.

4.5 Maintenance and 'Old Code'

As software systems become older, the modular structure tends to degenerate. Import and export lists grow and rarely shrink, even though an imported item is no longer used in the importing module and an exported item is no longer used outside the module, or even inside the module. SMILE assists users in restructuring old systems by moving items from one module to another and adjusts the imports and exports accordingly, by adjusting the imports and exports throughout the database to reflect the actual interconnections determined by cross-references, and by detecting unused items.

SMILE provides facilities to bring externally developed 'old code' into a database, so it can assist future maintenance and enhancement activities. SMILE can also copy modules from one database to another. SMILE makes it easy to use software maintained outside of a SMILE

[4]This performance problem is reduced when SMILE runs in a distributed workstation environment; see Section 5

database: Every module and every database may have a *prelude*, which lists external files and definitions of outside procedures; the corresponding object code files and Unix libraries are listed in SMILE library items. The add, remove, and change commands apply to libraries, as do the browsing facilities. The names of necessary libraries are given as arguments to the build command to incorporate them in an executable system. SMILE helps users create new Unix archives and libraries. It can produce a Unix archive from the C items in the database and can generate a single object code file that can be used as a library outside SMILE or within other SMILE databases.

5 Implementation

SMILE was originally called IPC, for Incremental Program Constructor, but the name was soon changed to SMILE. A prototype implementation was written in the Unix shell language during August 1979; it was used in September 1979 to bootstrap to a more advanced implementation in GC. These two versions ran on a PDP™ 11/70 under Unix Version 7.

SMILE was soon ported to a VAX (both 750 and 780) under Berkeley Unix, where it supported the intensive Gandalf prototype [10] implementation in 1980 and 1981 and the development and maintenance of the production-quality Gnome environment starting in 1982. SMILE was ported to the Sun Workstation™ in 1984 and to the MicroVAX™ workstation in 1985. The MicroVAX version is distributed by virtue of the Mach variant of Unix 4.3 BSD. The current implementation consists of 15,000 lines of GC source code, which is available on request from the Gandalf project at CMU.

Details

Although the original IPC was implemented in the Unix shell language, neither IPC nor the later versions of SMILE should be thought of simply as user shells. SMILE maintains its own database of all information about a software project and provides its own commands for carrying out development and maintenance activities; in effect, SMILE presents its own model of the programming process.

SMILE maps its database onto the Unix file system in a hierarchical manner. Each database corresponds to a directory, which contains a subdirectory for each project, which in turn contains a subdirectory for each module. Each module directory contains two files listing the imports and exports, respectively, and four subdirectories, one each for procedures, variables, types, and macros. The text of each item is stored in a separate file. This mapping to the file system is not visible to users. Cross-referencing information, status, and other derived attributes are maintained in a graph structure. This graph is dumped in binary form to a file within the database to persist between invocations of SMILE. A backup copy of the graph is also maintained, but if both the original and backup are corrupted, the graph can be regenerated from the database.

SMILE protects its users from operating system crashes, which might leave a database in an inconsistent state. SMILE automatically checks its database at the beginning of every session: if derived information such as error messages or object code has been lost, SMILE resets status information to make sure they are rederived. If the most recent session with this database was done using a previous version of SMILE itself, SMILE automatically reformats the graph structure and the database and adds default values for any new kinds of attributes. Approximately 30% of SMILE's source code is for disaster recovery and self repairs.

SMILE hides the Unix file system and its tools and utilities from its users, with the exception that it calls the user's favorite text editor. The default text editor at CMU is Emacs [9], but a different default can be substituted at each site. SMILE invokes lint to detect static semantic errors in source code objects, cc to compile modules, and make to generate executable systems. The variants of grep support SMILE's searches through source text and other objects.

We have not modified these tools; instead, SMILE automatically transforms objects into the format required by each tool. For example, SMILE combines the items of a module into a single file in the correct order for input to the compiler. This made it easy to port SMILE from one version of Unix to another and to use new tools as they became available (*e.g.*, lint replaced cc for static semantic analysis in 1982) without these changes being visible to the users. We believe it would not be difficult to port SMILE to a non-Unix operating system, providing it supplied similar tools; the only local tools that are mandatory are a text editor, a C compiler, and a linker.[5]

6 Related Systems

SMILE is similar to knowledge-based programming environments, advanced programming language environments, language-based editors and software engineering environments. In the following paragraphs, we describe the advantages and disadvantages of these systems with respect to SMILE.

Knowledge-Based Environments: The CommonLisp Framework (CLF) [6], Refine™ [22] and other knowledge-engineering environments can provide SMILE-like automation via condition/action rules [5]. However, they cannot recognize the alternative results of actions, *e.g.*, the compiler may terminate successfully, producing object code, or unsuccessfully, producing error messages. None of these environments support multiple simultaneous users. On the other hand, SMILE is not extensible, so it is not as easy to add new kinds of objects and new tools.

Language Environments: Advanced programming languages such as Interlisp [26], Loops [23] and Smalltalk-80™ [8] include run time environments that are indistinguishable from single-user programming environments. Although they provide SMILE-like facilities, these are strongly tied to the programming language. The implementation of SMILE is specific to the GC, but it would not be very difficult to reimplement for another conventional programming language,

[5]Early versions of SMILE used only the tools and utilities provided by Unix, but recent versions also include special tools for language-oriented programming environments. These tools are not relevant to the facilities described in this paper.

provided corresponding tools were available. However, language environments can integrate debugging facilities with the other tools.

Language-Based Environments:[8] Language-based environments add many of the advantages of language environments to conventional languages such as Pascal. The Synthesizer [25] and Pecan [19] are examples of specific environments, while the Synthesizer Generator [20] and Gandalf are systems for generating such environments from formal descriptions. Most language-based environments provide advanced user interfaces with menus and pointing devices, and perform various activities in response to programmer actions, but they are unable to anticipate the potential results of actions and warn users before the damage is done. The practicality of these environments is limited, since the entire software system is maintained as a single abstract syntax tree; further, it is difficult to incorporate existing programming tools into these environments.

Software Engineering Environments: SMILE is most similar to Cedar [27], DSEE™ [14], Arcadia [24] and other large-scale environments for software development and maintenance. Like SMILE, these environments provide an interface between programming tools and the user on the one hand, and between programming tools and the software database on the other. Such environments typically provide more advanced version control and project management facilities than SMILE, but they leave individual programmers to the standard edit/compile/debug cycle supported by traditional tools.

7 Conclusions

SMILE's primary contribution is the apparently intelligent assistance that spans both the activities of individual programmers and the coordination of multiple programmers. SMILE provides this assistance by

- maintaining all information about a software project in a database;
- integrating Unix tools into a new model of development and maintenance that hides the particularities of Unix tools;
- actively participating in the development and maintenance processes by deriving data when possible from previously stored information, automating the invocation of these tools and anticipating the consequences of tool processing;
- imposing a structure on software development activities that permits it to 'know' what the programmers are doing at all times, to 'infer' what they are likely to do next, and to 'judge' what it can appropriately do for them;
- recovering from external and internal failures and repairing its databases automatically, making it sufficiently robust and reliable for production use.

SMILE provides this assistance without a knowledgebase of rules describing the software development process. Instead, certain 'common sense' about software development activities has been

[8]We use the term 'language-based environment' synonymously with 'language-based editor', 'structure-oriented environment', 'structure editor-based environment', 'syntax-directed editor', etc.

programmed directly into the environment, resulting in a production-quality intelligent assistant that several projects have relied on to develop and maintain their software.

Acknowledgements

In addition to the authors, other past and present members of the Gandalf Project at Carnegie-Mellon University have been active in SMILE's evolution over the past seven years. Raul Medina-Mora and David Notkin were involved in the early development. Barbara Denny, Bob Ellison and Charlie Krueger have been responsible for maintaining and enhancing SMILE at various times. Many other people have developed tools that are presently incorporated into SMILE. Nico Habermann is the principal investigator of the Gandalf Project.

References

[1] Robert Balzer.
A 15 Year Perspective on Automatic Programming.
IEEE Transactions on Software Engineering SE-11(11):1257-1268, November, 1985.

[2] Robert M. Balzer.
Living in the Next Generation Operating System.
In *Proceedings of the 10th World Computer Congress (IFIP Congress '86).* Dublin, Ireland, September, 1986.
To appear as a book published by Springer-Verlag.

[3] David R. Barstow and Howard E. Shrobe.
From Interactive to Intelligent Programming Environments.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984, pages 558-570.

[4] David R. Barstow, Howard E. Shrobe and Erik Sandewall.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[5] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin.
Programming Expert Systems in OPS5.
Addison-Wesley Publishing Co., Reading, MA, 1985.

[6] CLF Project.
Introduction to the CLF Environment.
March, 1986.
USC Information Sciences Institute.

[7] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices,* 19(5), May, 1984.

[8] Adele Goldberg and David Robson.
Smalltalk-80 The Language and its Implementation.
Addison-Wesley Publishing Co., Reading, MA, 1983.

[9] James A. Gosling.
Unix Emacs
Carnegie-Mellon University, Department of Computer Science, 1982.

[10] Gail E. Kaiser, Robert J. Ellison, David B. Garlan, David S. Notkin and Steven Popovich.
Gandalf User Manual and Tutorial.
In *The Second Compendium of Gandalf Documentation.* Carnegie-Mellon University, Department of Computer Science, 1982.

[11] Beverly L. Kedzierski.
Knowledge-Based Project Management and Communication Support in a System Development Environment.
In *Proceedings of the 4th Jerusalem Conference on Information Technology.* Jerusalem, Israel, May, 1984.

[12] Brian W. Kernighan and John R. Mashey.
The UNIX Programming Environment.
Software — Practice and Experience 9(1), January, 1979.
Appears in IEEE Computer, 12(4), April 1981 and in [4].

[13] Charlie Krueger.
Private communication.
August, 1986
Regarding largest system (ALOE) maintained in SMILE.

[14] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June, 1985.

[15] John R. Nestor.
Toward a Persistent Object Base.
In *Proceedings of the IFIP WG 2.4 International Workshop on Advanced Programming Environments*. June, 1986.
To appear as a book published by Springer-Verlag.

[16] David Notkin.
The GANDALF Project.
The Journal of Systems and Software 5(2):91-105, May, 1985.

[17] Dewayne E. Perry.
Position Paper: The Constructive Use of Module Interface Specifications.
In *Third International Workshop on Software Specification and Design*. London, England, August, 1985.

[18] C.V. Ramamoorthy, Vijay Garg and Rajeev Aggarwal.
Environment Modelling and Activity Management in Genesis.
In *Proceedings of SoftFairII: 2nd Conference on Software Development Tools, Techniques, and Alternatives*, pages 2-10. December, 1985.

[19] Steven P. Reiss.
Graphical Program Development with PECAN Program Development Systems.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[20] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[21] B. A. Sheil.
Power Tools for Programmers.
Datamation Magazine , 1983.
Reprinted in [4].

[22] Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold.
Research on Knowledge-Based Software Environments at Kestrel Institute.
IEEE Transactions on Software Engineering SE-11(11):1278-1295, November, 1985.

[23] Mark Stefik and Daniel G. Bobrow.
Object-Oriented Programming: Themes and Variations.
AI Magazine 6(4):40-62, Winter, 1986.

[24] Richard N. Taylor, Lori Clarke, Leon J. Osterweil, Jack C. Wiledon and Michal Young.
Arcadia: A Software Development Environment Research Project.
In *2nd International Conference on Ada Applications and Environments.* IEEE Computer Society, Miami Beach, FL, April, 1986.

[25] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9), September, 1981.
Reprinted in [4].

[26] Warren Teitelman and Larry Masinter.
The Interlisp Programming Environment.
IEEE Computer 14(4):25-34, April, 1981.
Reprinted in [4].

[27] Warren Teitelman.
A Tour Through Cedar.
IEEE Software 1(2):44-73, April, 1984.
Also appears in Proceedings of the Seventh International Conference on Software Engineering, 1984.

[28] Walter F. Tichy.
Software Development Control Based on Module Interconnection.
In *4th International Conference on Software Engineering.* September, 1979.

[29] Terry Winograd.
Breaking the Complexity Barrier (Again).
In *Proceedings of the ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages — Information Retrieval,* pages 13-30. Gaithersburg, MD, November, 1973.
Reprinted in [4].

AD-A182 094

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNLIMITED, UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT UNCLASSIFIED, UNLIMITED, DTIC, NTIS
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-86-TM-14	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-86-221
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. — 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARENGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JPO — 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628 85 C 0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63752F	N/A	N/A	N/A

11. TITLE (Include Security Classification)
INTELLIGENT ASSISTANCE WITHOUT ARTIFICIAL INTELLIGENCE

12. PERSONAL AUTHOR(S)
GAIL KAISER AND PETER FEILER

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM ... TO ...	SEPTEMBER 1986	18

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

SMILE IS A DISTRIBUTED, MULTI-USER SOFTWARE ENGINEERING ENVIRONMENT THAT BEHAVES AS AN INTELLIGENT ASSISTANT. SMILE PRESENTS A 'FILELESS ENVIRONMENT', DERIVES AND TRANSFORMS DATA TO SHELTER USERS FROM ENTERING REDUNDANT INFORMATION, AUTOMATICALLY INVOKES PROGRAMMING TOOLS, AND ACTIVELY PARTICIPATES IN THE SOFTWARE DEVELOPMENT AND MAINTENANCE PROCESS. UNLIKE OTHER INTELLIGENT ASSISTANTS, SMILE IS NOT A RULE-BASED ENVIRONMENT: ITS KNOWLEDGE OF SOFTWARE OBJECTS AND THE PROGRAMMING PROCESS IS HARDCODED INTO THE ENVIRONMENT. WE DESCRIBE SMILE'S FUNCTIONALITY AND EXPLAIN HOW WE ACHIEVED THIS FUNCTIONALITY WITHOUT RELIANCE ON ARTIFICIAL INTELLIGENCE TECHNOLOGY.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☒ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED, DTIC, NTIS
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 — 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE